APPENDIX A

```
PROCEDURE Create_Root_Node (root_ptr, node_value)
BEGIN
   Allocate_Node (root_ptr, node);
   Increment (node_counter);
   node[root_ptr].counter := node_counter;
   node[root_ptr].value := node_value;
   node[root_ptr].sibling_pointer := nil;
   node[root_ptr].child_pointer := nil;
   node[root_ptr].parent_pointer := nil;
END.
```

APPENDIX B

```
PROCEDURE Insert_Node (parent_ptr, node_value);
BEGIN
   INTEGER node_ptr, ptr;
   Allocate_Node (node_ptr, node);
   Increment (node_counter);
   node[node_ptr].counter := node_counter;
   node[node_ptr].value := node_value;
   ptr := node[parent_ptr].child_pointer;
   node[parent_ptr].child_pointer := node_ptr;
   node[node_ptr].sibling_pointer := ptr;
   node[node_ptr].child_pointer := nil;
   node[node_ptr].parent_pointer := parent_ptr;
END.
```

APPENDIX C1

```
PROCEDURE Delete_Node (node_ptr);
% This version of Delete_Node requires node to be deleted have no offspring.
BEGIN
  IF node[node_ptr].child_pointer = nil THEN
  BEGIN
    node[node[node_ptr].parent_pointer].child_pointer :=
      node[node_ptr].sibling_pointer;
    Deallocate_Node (node_ptr);
  END ELSE
    RETURN ("error: cannot delete node while offspring exist")
END.
```

APPENDIX C2

```
PROCEDURE Delete_Node (node_ptr);
% This version of Delete_Node deletes the node and its entire offspring.
BEGIN
  PROCEDURE Delete_Subtree (ptr);
  BEGIN
    IF node[ptr].child_pointer <> nil THEN
      Delete_Subtree (node[ptr].child_pointer);
    IF node[ptr].sibling_pointer <> nil THEN
      Delete_Subtree (node[ptr].sibling_pointer);
    Deallocate_Node (ptr);
  END;

  IF node[node_ptr].child_pointer <> nil THEN
    Delete_Subtree (node[node_ptr].child_pointer);
  node[node[node_ptr].parent_pointer].child_pointer :=
    node[node_ptr].sibling_pointer;
  Deallocate_Node (node_ptr);
END.
```

APPENDIX D

```
PROCEDURE Preorder_Traverse_Tree (node_ptr);
BEGIN
   DISPLAY (node[node_ptr].node_value);
   IF node[node_ptr].child_pointer <> nil THEN
      Preorder_Traverse_Tree (node[node_ptr].child_pointer);
   IF node[node_ptr].sibling_pointer <> nil THEN
      Preorder_Traverse_Tree (node[node_ptr].sibling_pointer);
END.
```

APPENDIX E

```
PROCEDURE Postorder_Traverse_Tree (node_ptr);
% Postorder traversal of a general tree is equivalent to inorder traversal of the
% binary tree that represents that general tree.
BEGIN
   IF node[node_ptr].child_pointer <> nil THEN
      Postorder_Traverse_Tree (node[node_ptr].child_pointer);
   DISPLAY (node[node_ptr].node_value);
   IF node[node_ptr].sibling_pointer <> nil THEN
      Postorder_Traverse_Tree (node[node_ptr].sibling_pointer);
END.
```

APPENDIX F

```
PROCEDURE Find_First_Node (ext_ptr, ext_nodes, ext_pointers, level,
int_pointers);
BEGIN
  ARRAY   ancestor_nodes[0:maxlevels];
  INTEGER save_level;
  BOOLEAN seeking;

  % find depth of tree
  level := -1;
  ptr := ext_node_ptr;
  WHILE ptr <> nil DO
  BEGIN
    level := *+1;
    ptr := ext_node_array[ptr].parent_ptr;
  END;
  save_level := level;

  % retrieve specified lineage
  IF level >= 0 THEN
  BEGIN
    ancestor_nodes[level+1].counter := max_int;
    ptr := ext_node_ptr;
    WHILE ptr <> nil DO
    BEGIN
      ancestor_nodes[level] := ext_nodes[ptr];
      level := *-1;
      ptr := ext_node_array[ptr].parent_ptr;
    END:
    level := 0;
    int_pointers[level] := root_ptr;
  END;

  % establish continuation lineage (setup simulated recursion stack)
  seeking := TRUE;
  WHILE seeking DO
  BEGIN
    IF level < 0 THEN
    BEGIN  % at end of tree or no start pointer - start at root
      seeking := FALSE;
      level := 0;
      int_pointers[level] := root_ptr;
    END ELSE
    IF int_pointers[level] = nil THEN
```

```
      BEGIN % no nodes at this level - drop back level and get next sibling
        level := *-1;
        IF level >= 0 THEN
           int_pointers[level] := node[int_pointers[level]].sibling_ptr;
      END ELSE
      IF (node[int_pointers[level]].counter > ancestors[level].counter) THEN
      BEGIN % already visited this node - get next sibling
        int_pointers[level] := node[ptr].sibling_ptr;
      END ELSE
      IF (node[int_pointers[level]].counter = ancestors[level].counter) THEN
      BEGIN % node exists - increase level and get child
        level := *+1;
        int_pointers[level] := node[int_pointers[level-1]].child_ptr;
      END ELSE
      BEGIN % found first node at this level not yet visited
        seeking := FALSE;
      END;
    END;
  END;
```

APPENDIX G

```
INTERFACE (Single_Step_Preorder_Traverse_Tree,
        Partial_Preorder_Traverse_Tree);


PROCEDURE Single_Step_Preorder_Traverse_Tree (ext_node_ptr, ext_nodes);
BEGIN
   Find_First_Node (ext_node_ptr, ext_nodes,
              level, int_pointers);
   Insert_First_Lineage (ext_node_ptr, ext_ptr, ext_nodes, ext_pointers,
                level, int_pointers);
END:


PROCEDURE Partial_Preorder_Traverse_Tree (ext_node_ptr, ext_nodes);
BEGIN
   INTEGER ext_ptr, level;
   BOOLEAN finished;
   ARRAY   int_pointers,ext_pointers[0:maxlevels];

   Find_First_Node (ext_node_ptr, ext_nodes,
              level, int_pointers);
   Insert_First_Lineage (ext_node_ptr, ext_ptr, ext_nodes, ext_pointers,
                level, int_pointers);
   finished := FALSE;
   WHILE NOT finished DO
   BEGIN
      Find_Next_Node (level, int_pointers, ext_pointers);
      IF level >= 0 THEN
      BEGIN
         Insert_Next_Node (ext_ptr, ext_nodes, ext_pointers,
                   level, int_pointers);
         IF ext_ptr+1 = SIZE (ext_nodes) THEN
            finished := TRUE;  % no more nodes fit into external tree
      END
      ELSE
         finished := TRUE;  % traversal of the tree finished
   END;
END;
```

APPENDIX H

```
PROCEDURE Find_Next_Node (level, int_pointers, ext_pointers);
BEGIN
  % simulate recursion - traverse child subtree, then traverse sibling subtree
  level := *+1;
  int_pointers[level] := node[int_pointers[level-1]].child_ptr;
  WHILE (level >= 0) CAND
     (int_pointers[level] = nil) DO
  BEGIN
    ext_pointers[level] := nil;
    level := *-1;
    IF level => 0 THEN
      int_pointers[level] := node[int_pointers[level]].sibling_ptr;
  END;
END;
```

APPENDIX I

```
PROCEDURE Insert_First_Lineage (ext_node_ptr, ext_ptr, ext_nodes,
ext_pointers, level, int_pointers);
BEGIN
  ext_ptr := 0;
  ext_nodes[ext_ptr] := node[int_pointer[ext_ptr]];
  WHILE ext_ptr < level DO
  BEGIN
    ext_ptr := *+1;
    ext_node[ext_ptr] := node[int_pointer[ext_ptr]];
    % fix links; set non-nil sibling pointers to "inuse" flag value
    IF ext_node[ext_ptr].sibling_ptr <> nil THEN
      ext_node[ext_ptr].sibling_ptr := inuse;
    ext_node[ext_ptr-1].child_ptr := ext_ptr;
    ext_node[ext_ptr].parent_ptr := ext_ptr-1;
    ext_pointers[ext_ptr] := ext_ptr;
  END;
  % set non-nil child pointer to "inuse" flag value
  IF ext_node[ext_ptr].child_ptr <> nil THEN
    ext_node[ext_ptr].child_ptr := inuse;
  ext_node_ptr := ext_ptr;
END;
```

APPENDIX J

```
PROCEDURE Insert_Next_Node (ext_ptr, ext_nodes, ext_pointers, level,
int_pointers);
BEGIN
   ext_ptr := *+1;
   ext_node[ext_ptr] := node[int_pointers[level];
   % fix parent's link or previous sibling's link to point to this node
   IF ext_pointers[level] = nil THEN
      ext_node[ext_pointers[level-1]].child_ptr := ext_ptr
   ELSE
      ext_node[ext_pointers[level]].sibling_ptr := ext_ptr;
   % set non-nil sibling pointer to "inuse" flag
   IF ext_node[ext_ptr].sibling_ptr <> nil THEN
      ext_node[ext_ptr].sibling_ptr := inuse;
   % set non-nil child pointer to "inuse" flag value
   IF ext_node[ext_ptr].child_ptr <> nil THEN
      ext_node[ext_ptr].child_ptr := inuse;
   % set parent link
   ext_node[ext_ptr].parent_ptr := ext_pointers[level-1];
   ext_pointers[ext_ptr] := ext_ptr;
END;
```

United States Patent & Trademark Office

Office of Initial Patent Examination -- Scanning Division

SCANNED # 8

Application deficiencies found during scanning:

□ Page(s)____________ of ______________________________ were not present for scanning.
(Document title)

□ Page(s)____________ of ______________________________ were not present for scanning.
(Document title)

Pages 25-32 of the specification are appendix.

□ ***Scanned copy is best available.***